Project Advice

Adapted from notes by
Christopher Manning

Organization

- Work in teams 1-3, fostering cooperation and exploring more solutions
- Exploit deep learning frameworks (like those presented in the course)
- You can use pre-trained models, but you should build your own architecture on top of them

Project Planning

- Choose the task you want to perform
- Find a relevant (key) research paper for your topic
- Find suitable dataset for training your system and an evaluation metrics to use

Final Report Structure

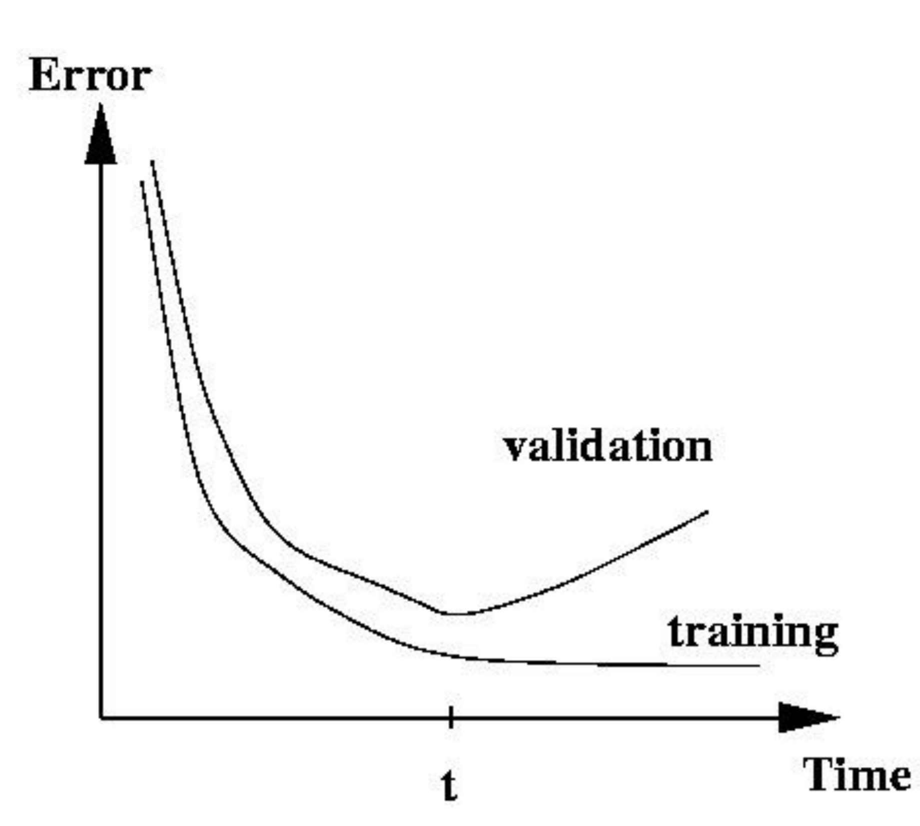
- **Introduction**. Describe the task
- **Prior work**. Provide a summary of literature on the subject and what you took as inspiration
- **Data**. Describe the dataset you have used and the evaluation metrics
- **Architecture**. Describe the architecture(s) of the model(s) you have implemented
- **Experiments**. Describe the experiments you carried out
- **Results and Analysis**. Evaluate the results and compare them to the SotA
  - Optional: carry out error analysis, ablation studies and critical comparisons with alternative solutions
- **Conclusions**. Draw conclusions and provide suggestions for improvements

Use of datasets

- Many publicly available datasets are released with a **train/dev/test** structure. **We're all on the honor system to do test-set runs only when development is complete.**
- Splits like this presuppose a fairly large dataset.
- If there is no dev set or you want a separate tune set, then you create one by splitting the training data
  - compromise against the reduction in train-set size
  - Cross-validation might help
- Having a fixed test set ensures that all systems are assessed against the same gold data. This is generally good, but:
  - It is problematic where the test set turns out to have unusual properties that distort progress on the task.
  - It doesn’t give any measure of variance.
  - It’s only an unbiased estimate of the mean if only used once.

Dealing with overfit

- When training, models **overfit** to what you are training on
  - The model correctly describes what happened to occur in particular data you trained on, but the patterns are not general enough patterns to be likely to apply to new data
- The way to avoid problematic overfitting (lack of generalization) is using **independent** validation and test sets ...



Don't touch the test set

- You build (estimate/train) a model on a **training set**.
- Often, you then set further hyperparameters on another, independent set of data, the **tuning/dev set**
  - The tuning set is the training set for the hyperparameters!
- You measure progress as you go on a **dev set** (development test set or validation set)
  - If you do that a lot you overfit to the dev set so it can be good to have a second dev set, the **dev2** set
- **Only at the end**, you evaluate and present final numbers on a **test set**
  - Use the final test set **extremely** few times ... ideally only once

Separate datasets

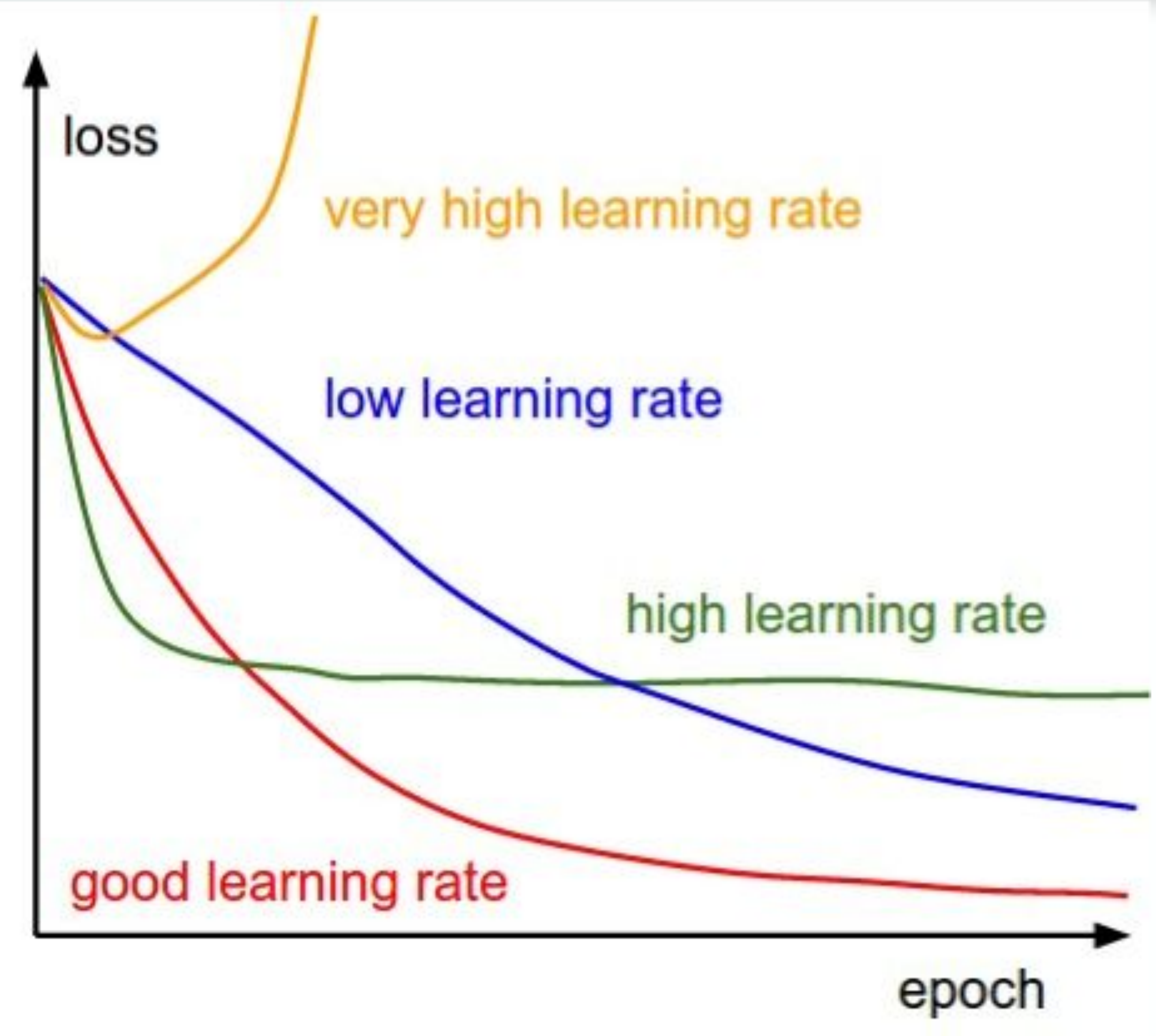
- The **train**, **tune**, **dev**, and **test** sets need to be completely distinct
- It is invalid to test on material you have trained on
  - You will get a falsely good performance. We usually overfit on train
- You need an independent tuning set
  - The hyperparameters won't be set right if tune is same as train
- If you keep running on the same evaluation set, you begin to overfit to that evaluation set
  - Effectively you are "training"on the evaluation set...you are learning things that do and don't work on that particular eval set and using the info
- To get a valid measure of system performance you need another untrained on, **independent** test set ... hence dev2 and final test

Training Experiments

- Start with a positive attitude!
  - **Neural networks want to learn!**
  - If the network isn't learning, you're doing something to prevent it from learning successfully
- Realize the grim reality:
  - **There are lots of things that can cause neural nets to not learn at all or to not learn very well**
  - Finding and fixing them ("debugging and tuning") can often take more time than implementing your model
- It's hard to work out what these things are
  - But experience, experimental care, and rules of thumb help!

Models are sensitive to learning rates

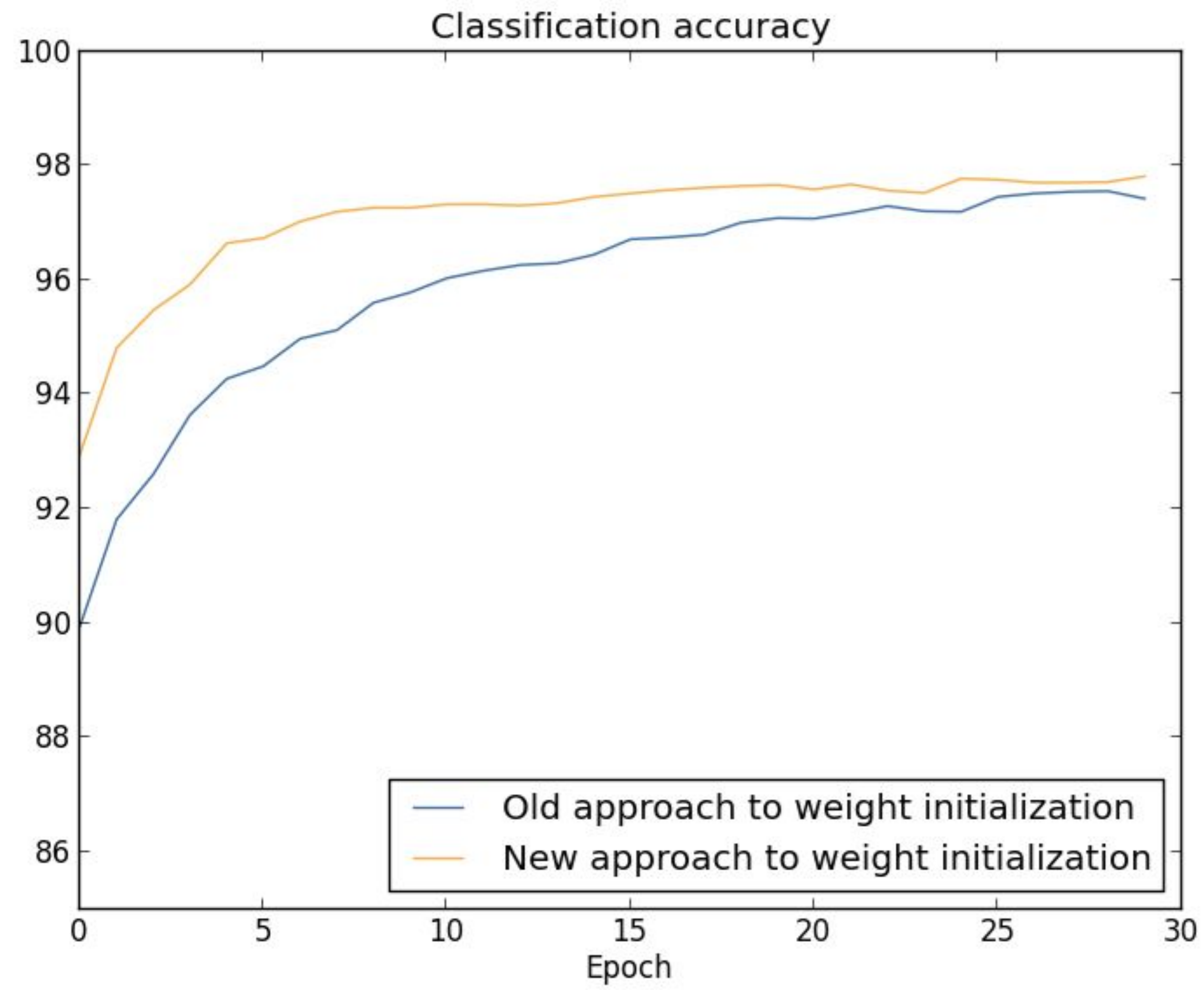
Adaptive optimers address this:
Adagrad, Adam, AdamW



From Andrej Karpathy, CS231n course notes

Models are sensitive to initialization

Pre-trained models address this



From Michael Nielsen
http://neuralnetworksanddeeplearning.com/chap3.htm

Use Transformers

- Transformers embed much linguistic knowledge that they can be often fine-tuned to perform other tasks
- HuggingFace provides a repository of a variety of pre-trained models and a uniform interface to use them https://huggingface.co/models
- Focus on:
  - fine-tuning
  - domain adaptation followed by fine-tuning
  - zero- few-shot learning
  - Prompt-tuning

Training a gated RNN

1. Use an LSTM or GRU: *it makes your life much simpler!*
2. Initialize recurrent matrices to be orthogonal
3. Initialize other matrices with a sensible (**small!**) scale
4. Initialize forget gate bias to 1: *default to remembering*
5. Use adaptive learning rate algorithms: *Adam, AdaDelta, ...*
6. Clip the norm of the gradient: *1–5 seems to be a reasonable threshold when used together with Adam or AdaDelta.*
7. Either only dropout vertically or look into using Bayesian Dropout (Gal & Gahramani)
8. *Be patient! Optimization takes time*

Experimnetal strategy

- **Work incrementally!**
- Start with a very simple model and get it to work!
  - It's hard to fix a complex but broken model
- Add bells and whistles one-by-one and get the model working with each of them (or abandon them)
- **Initially run on a tiny amount of data**
  - You will see bugs much more easily on a tiny dataset
  - Something like 4–8 examples is good
  - Often synthetic data is useful for this
  - Make sure you can get 100% on this data
  - Otherwise your model is definitely either not powerful enough or it is broken

Experimental strategy

- Run your model on a large dataset
  - It should still score close to 100% on the training data after optimization
  - Otherwise, you probably want to consider a more powerful model
  - Overfitting to training data is **not** something to be scared of when doing deep learning
  - These models are usually good at generalizing because of the way distributed representations share statistical strength regardless of overfitting to training data
- But, still, you now want good generalization performance:
  - Regularize your model until it doesn't overfit on dev data
    - Strategies like L2 regularization can be useful (Keras)

```
model.add(Dense(64, input_dim=64,
                kernel_regularizer=regularizers.l2(0.01)
```

    - But normally **generous dropout** is the secret to success

Details matter!

- Be very familiar with your (train and dev) data, don't treat it as arbitrary bytes in a file!
  - Look at your data, collect summary statistics
  - Look at your model's outputs, do error analysis
- Tuning hyperparameters is **really** important to almost all of the successes of NNets

Good luck with your projects!